ENGR 76: Information Science and Engineering Winter 2026 — Lectures 2–4

Source Coding and Lossless Compression

Siddharth Chandak

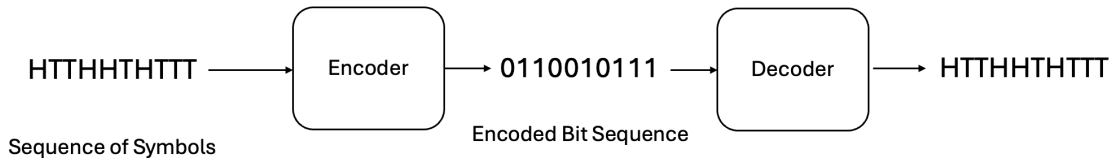
We begin this course by studying source coding (lossless compression): how to represent sequences of symbols using as few bits as possible while still allowing perfect reconstruction. We introduce codes and decoding guarantees, develop Huffman coding as an optimal prefix-free method for minimizing average code length, and then use probability and entropy to understand the fundamental limits of compression. We show that entropy characterizes the minimum achievable number of bits per symbol for independent and identically distributed sources, and we study how block coding allows us to approach this limit arbitrarily closely. We then turn to dependent sources and discuss how exploiting structure through suitable transformations can lead to further compression.

1 Symbols, Alphabets, and Codes

An *alphabet* $\mathcal{X}$ is a finite set of symbols:

$$\mathcal{X} = \{x_1, x_2, \ldots, x_M\},$$

where M denotes the number of symbols. A *code* is a mapping which assigns to each symbol $x \in \mathcal{X}$ a finite binary string $c(x)$, called its *codeword*. The length of the codeword $c(x)$ is denoted by $\ell(x)$. Given a code c, the *encoder* maps a sequence of symbols to a binary string (encoded bit sequence or encoded bitstream) by concatenating the corresponding codewords. The *decoder* is a mapping that takes a bitstream produced by the encoder and reconstructs the original sequence.



Example 1. *Let $\mathcal{X} = \{H, T\}$ represent the outcomes of a coin flip. Here $M = 2$. Consider the code*

$$H \mapsto 0, \qquad T \mapsto 1.$$

For the sequence

$$HTTHHTHTTT,$$

the encoded bitstream or the encoded bit sequence is

$$0110010111.$$

2 Fixed-Length Codes

In this section, we study the simplest class of codes, namely those in which all codewords have the same length. These are referred to as *fixed-length codes*. In such codes, every symbol in the alphabet is represented using exactly the same number of bits. This uniform structure makes both encoding and decoding straightforward, since the decoder can simply divide the received bitstream into blocks of equal size.

Definition 1 (Fixed-Length Code)**.** *A code c is called a fixed-length code if there exists an integer k such that*

$$\ell(x) = k \quad \textit{for all } x \in \mathcal{X}.$$

Let M be the number of symbols in the alphabet. Since there are 2^k distinct binary strings of length k, a fixed-length code with distinct codewords must satisfy

$$2^k \geq M.$$

Thus, the minimum possible codeword for a fixed-length code with distinct codewords is

$$k = \lceil \log_2 M \rceil.$$

Here $\lceil y \rceil$ denotes the ceiling function which outputs the smallest integer greater than or equal to y.

Example 2. *Let $\mathcal{X} = \{A, B, C\}$, so $M = 3$. Since $\lceil \log_2 3 \rceil = 2$, we may use the fixed-length code*

$$A \mapsto 00, \quad B \mapsto 01, \quad C \mapsto 10,$$

Each symbol is represented using exactly two bits.

2.1 Decoding Fixed-Length Codes

It is very straightforward to decode fixed-length codes using the following algorithm.

Decoding Algorithm:

- Invert the codebook to obtain a mapping from codewords to symbols.
- Divide the encoded bitstream into consecutive blocks of length k.
- Replace each block by its corresponding symbol.

The following example illustrates this decoding algorithm.

Example 3. *Consider the fixed-length code with $k = 2$ given by*

$$A \mapsto 00, \quad B \mapsto 01, \quad C \mapsto 10.$$

Suppose the encoded bitstream is 00011010*. We divide the bitstream into consecutive blocks of length* 2*:*

$$00\ 01\ 10\ 10.$$

Replacing each block by its corresponding symbol, we obtain the decoded sequence $A\,B\,C\,C$.

Fixed-length codes are easy to decode but do not exploit differences in symbol frequencies and can therefore be inefficient when some symbols occur more frequently than others. In many practical sources, some symbols appear much more often than others. If we use the same number of bits for every symbol, we end up using unnecessarily many bits to represent the frequent symbols. Intuitively, we would like to assign shorter codewords to more frequent symbols and longer codewords to less frequent ones in order to reduce the average number of bits per symbol. This motivates the study of variable-length codes, and in particular prefix-free codes, which allow different codeword lengths while still guaranteeing reliable decoding.

3 Desirable Properties of Codes

We now study several important properties that a code may satisfy. These properties impose increasingly stronger constraints to ensure reliable decoding of sequences of symbols. For the first two properties, we begin by defining the property and then provide examples illustrating why additional, stronger properties are necessary.

3.1 Non-Singular Codes

Before introducing any formal definitions, let us first consider what can go wrong when assigning codewords to symbols. At a minimum, we would expect that different symbols should not be assigned the same codeword. Otherwise, even a single transmitted symbol cannot be recovered uniquely.

Example 4. *Let $\mathcal{X} = \{A, B, C, D\}$ and consider the code*

$$A \mapsto 0, \quad B \mapsto 1, \quad C \mapsto 0, \quad D \mapsto 1.$$

Here multiple symbols share the same codeword. For instance, both A and C are mapped to 0. Thus, upon receiving the bit 0, we cannot determine which symbol was sent. This code is therefore unacceptable even at the single-symbol level.

The previous example motivates the following basic requirement.

Definition 2 (Non-Singular Code)**.** *A code is called non-singular if distinct symbols are assigned distinct codewords:*

$$x \neq x' \quad \Rightarrow \quad c(x) \neq c(x').$$

Non-singularity guarantees that individual symbols are distinguishable. However, this condition alone is still not sufficient to ensure reliable decoding of entire sequences, since ambiguity may arise when codewords are concatenated.

Example 5. *Consider the code*

$$A \mapsto 0, \quad B \mapsto 00, \quad C \mapsto 1, \quad D \mapsto 11.$$

This code is non-singular, since every symbol has a distinct codeword. However, the bitstream 00 can be interpreted either as the single symbol B or as the sequence AA. Thus, although individual codewords are distinct, sequences of symbols may still be decoded ambiguously.

We therefore conclude that while non-singularity is necessary, it is not sufficient for reliable sequence-level decoding. This motivates the introduction of stronger properties that guarantee unique decodability of arbitrary symbol sequences.

3.2 Uniquely Decodable Codes

Non-singularity guarantees that individual symbols are distinguishable. However, what ultimately matters is whether entire sequences of symbols can be recovered unambiguously after encoding. This leads to a stronger requirement.

Definition 3 (Uniquely Decodable Code)**.** *A code is uniquely decodable if every finite sequence of symbols maps to a unique binary string under encoding. Equivalently, no encoded bitstream corresponds to two different symbol sequences.*

Thus, even after concatenating codewords, the original sequence must be recoverable in exactly one way. The following example illustrates that a code may be uniquely decodable even though decoding is not instantaneous.

Example 6. *Consider the code*

$$A \mapsto 0, \quad B \mapsto 01.$$

This code is uniquely decodable. When we read a 0*, we cannot immediately determine whether it represents the symbol* A *or is the beginning of the codeword* 01 *for* B*. We must read one additional bit: if the next bit is* 1*, we decode* B*; if it is* 0*, we decode* A *and continue.*

More generally, consider the code

$$A \mapsto 0, \quad B \mapsto 0^k 1,$$

where 0^k *denotes a sequence of* k *zeros. This code is also uniquely decodable. However, upon reading a* 0*, the decoder may need to examine up to* $k+1$ *bits before determining whether the transmitted symbol was* A *or* B*. Thus decoding is unique, but not instantaneous, since the decoder may need to look ahead several bits before deciding which symbol was transmitted.*

Unique decodability ensures correctness of sequence-level decoding, but it does not guarantee that symbols can be decoded immediately as they are read. In such codes, the decoder may need to wait and examine additional future bits before determining which symbol was transmitted. This delay can complicate real-time or streaming applications, increase decoding latency, and require additional buffering. Moreover, the decoding procedure may become highly specific to the particular structure of the code, making implementation less uniform and harder to generalize. It is therefore desirable to have codes that allow symbols to be decoded as soon as their codewords are completed, without looking ahead. This motivates an even stronger property that allows instantaneous decoding.

3.3 Prefix-Free Codes

To avoid the lookahead and code-specific decoding complexity associated with uniquely decodable codes, we introduce a stronger structural property that guarantees instantaneous decoding.

Definition 4 (Prefix-Free Code)**.** *A code is called prefix-free if no codeword is a prefix of any other codeword, i.e., no codeword appears at the beginning of another codeword.*

In other words, once a complete codeword has been read, it cannot be the beginning of a longer valid codeword.

Example 7. *The code*

$$A \mapsto 0, \quad B \mapsto 10, \quad C \mapsto 110, \quad D \mapsto 111$$

is prefix-free. None of the codewords is the prefix of any other, so each symbol can be identified as soon as its codeword is completed.

Example 8. *The code*

$$A \mapsto 0, \quad B \mapsto 01$$

is not prefix-free, since the codeword for A is a prefix of the codeword for B.

Prefix-free codes allow for *instantaneous decoding*: each symbol can be decoded as soon as its complete codeword is read, without examining future bits. This property follows directly from the definition. If no codeword is a prefix of another, then whenever the current buffer matches a valid codeword, it cannot possibly be extended into a longer codeword. Hence the decoder can safely output the corresponding symbol immediately. This idea is reflected in the following simple decoding procedure.

Decoding Algorithm (Prefix-Free Codes):

- Invert the codebook to obtain a mapping from codewords to symbols.
- Initialize an empty buffer `buf`.
- Read the encoded bitstream one bit at a time:
  - Append the next bit to `buf`.
  - If `buf` matches a codeword in the inverted codebook:
    - Output the corresponding symbol.
    - Reset `buf` to empty.
  - Otherwise, continue reading bits.
- Stop when all bits have been read.

Because no codeword is a prefix of another, this procedure never encounters ambiguity. As soon as a codeword is matched, it must correspond to exactly one symbol, and it cannot be part of a longer valid codeword. Therefore prefix-free codes are uniquely decodable.

Example 9. *Consider the prefix-free code*

$$A \mapsto 0, \quad B \mapsto 10, \quad C \mapsto 110, \quad D \mapsto 111.$$

Suppose the encoded bitstream is

$$01101101110.$$

We decode it sequentially as follows. We read the first bit 0*, which matches the codeword for A, so we output A and reset the buffer. Next we read* 110*, which matches C, so we output C. Then we read* 110 *again and output another C. Finally we read* 111 *and output D, followed by* 0 *which corresponds to A. Thus the decoded sequence is*

$$A\,C\,C\,D\,A.$$

At each step, as soon as a valid codeword is recognized, the corresponding symbol is immediately determined, and no lookahead beyond the current codeword is required.

Thus prefix-free codes provide both correctness at the sequence level and simplicity of decoding, making them particularly attractive in practice.

3.4 Relationship Between the Properties

The properties discussed above are related through the following chain of implications:

$$\text{Prefix-Free} \;\Rightarrow\; \text{Uniquely Decodable} \;\Rightarrow\; \text{Non-Singular}.$$

These implications reflect increasing strength of structural constraints. Every prefix-free code is uniquely decodable, since instantaneous decoding guarantees that no ambiguity can arise when concatenating codewords. Every uniquely decodable code must be non-singular, since if two symbols shared the same codeword, even a single symbol could not be decoded uniquely.

However, the converses do not hold in general. A non-singular code need not be uniquely decodable, and a uniquely decodable code need not be prefix-free. Thus prefix-free codes form a strictly smaller class of codes than uniquely decodable codes. At first glance, one might worry that restricting attention to prefix-free codes could limit performance. The following fact shows that this is not the case.

Fact 1. *For any uniquely decodable code for a given alphabet $\mathcal{X}$, there exists a prefix-free code for the same alphabet with the same codeword lengths for each symbol $x \in \mathcal{X}$.*

This result implies that when optimizing over codeword lengths (for example, when minimizing average code length), there is no loss in restricting attention to prefix-free codes. Any achievable set of codeword lengths for a uniquely decodable code can also be achieved by a prefix-free code. Therefore, in the study of optimal lossless coding, we may focus exclusively on prefix-free codes without sacrificing optimality.

4 Information Sources and Probability Review

So far, we have studied how to assign binary codewords to symbols and how to ensure reliable decoding. We next discuss how these sequences of symbols are generated. We refer to the mechanism that generates these symbols as a *source*.

Definition 5 (Source)**.** *A* source *generates a sequence of symbols*

$$X_1, X_2, X_3, \ldots,$$

where each X_i takes values in a finite alphabet $\mathcal{X}$.

We take a probabilistic viewpoint, where the source is modeled as a random variable. This allows us to describe the source using probabilities and to analyze coding performance in terms of average behavior over long sequences. Let us quickly review some basics of probability and random variables.

4.1 Random Variables and Distributions

Intuitively, a random variable is a mathematical object that represents the outcome of a random experiment.

Definition 6 (Discrete Random Variable)**.** *A discrete random variable X takes values in a finite alphabet $\mathcal{X} = \{x_1, \ldots, x_M\}$, where each value x_i occurs with probability $\mathbb{P}(X = x_i) = p(x_i)$ for $i = 1, \ldots, M$.*

Let us discuss two examples.

Example 10 (Fair coin flip)**.** *Let X denote the outcome of a fair coin flip with alphabet $\mathcal{X} = \{H, T\}$. Then $p(H) = \mathbb{P}(X = H) = 0.5$ and $p(T) = \mathbb{P}(X = T) = 0.5$.*

Example 11 (Fair die roll)**.** *Let X denote the outcome of a fair die roll with alphabet $\mathcal{X} = \{1, 2, 3, 4, 5, 6\}$. All outcomes are equally likely, so $p(k) = \mathbb{P}(X = k) = 1/6$ for $k = 1, \ldots, 6$.*

Intuitively, the probabilities $p(x)$ describe the long-run behavior of repeated independent experiments. For example, if a fair die is rolled many times, then although individual outcomes are unpredictable, the fraction of times each face appears will be close to $1/6$. Thus, probabilities can be interpreted as long-run empirical frequencies when the experiment is repeated a large number of times (this can be formally stated using *law of large numbers*, which is not required for this course).

We next discuss the axioms of probability, which are intuitive properties we expect from any probability distribution. In the discrete setting relevant for this course, these axioms are simple and directly connected to the examples above.

A1. For every $x \in \mathcal{X}$, $\quad 0 \le p(x) \le 1$.

A2. $\quad \sum_{x \in \mathcal{X}} p(x) = 1$.

A3. If $A_1, \ldots, A_n$ are disjoint events, then

$$\mathbb{P}\Big(\bigcup_{i=1}^{n} A_i \Big) = \mathbb{P}(A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_n) = \sum_{i=1}^{n} \mathbb{P}(A_i).$$

These axioms capture the basic requirements that any probability distribution must satisfy. The first axiom states that probabilities are numbers between 0 and 1, reflecting the fact that an event cannot be less than impossible or more than certain. The second axiom ensures normalization: the total probability assigned to all possible outcomes in the alphabet must equal 1. One way to interpret this is to think of probabilities as long-run fractions of occurrence. If we repeat an experiment many times, the fraction of times each outcome appears must add up to the whole, and hence the probabilities must sum to 1. The third axiom formalizes additivity for disjoint events. If events cannot occur simultaneously, then the probability that one of them occurs is simply the sum of their individual probabilities.

4.2 Independence

Intuitively, two random variables are independent if observing one does not change our beliefs about the other. In other words, knowing the value of X provides no information about the value of Y, and vice versa. The following definition formalizes this idea.

Definition 7 (Independence)**.** *Two random variables X and Y are independent if for all values x in the alphabet of X and y in the alphabet of Y,*

$$\mathbb{P}(X = x, Y = y) = \mathbb{P}(X = x)\, \mathbb{P}(Y = y).$$

4.3 Expectation

We define the expectation of a random variable next.

Definition 8 (Expectation)**.** *Let X be a discrete random variable taking real values in $\mathcal{X}$ with probabilities $p(\cdot)$. Its expectation (mean) is*

$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} p(x)\, x = p(x_1)x_1 + \ldots + p(x_M)x_M.$$

Intuitively, if we repeat the same experiment many times, the average of the observed outcomes will be close to the expectation, which is simply a weighted average of the values in the alphabet $\mathcal{X}$ with weights equal to their respective probabilities.

Example 12. *Let X be the outcome of a fair dice roll. Then,*

$$\mathbb{E}[X] = \sum_{k=1}^{6} k \cdot \frac{1}{6} = \frac{1+2+3+4+5+6}{6} = 3.5.$$

Similar to the above definition, we can also define the expectation of a function of random variable. Suppose $Z = g(X)$, then Z is also a random variable, and

$$\mathbb{E}[Z] = \mathbb{E}[g(X)] = \sum_{x \in \mathcal{X}} p(x)\, g(x).$$

5 The Problem of Source Coding

We model the output of the source as a random variable X taking values in a finite alphabet $\mathcal{X}$, with $\mathbb{P}(X = x) = p(x)$. A code assigns a binary codeword to each symbol $x \in \mathcal{X}$. Since the symbol produced by the source is random, the length of the corresponding codeword is also a random quantity. Our objective in source coding is to design a prefix-free code that minimizes the average number of bits required to represent symbols generated by the source.

5.1 Expected Code Length

Recall that a code assigns a codeword $c(x)$ to each $x \in \mathcal{X}$, with length $\ell(x)$. We define expected code length (average number of bits per source symbol) as follows.

Definition 9 (Expected (Average) Code Length)**.** *For a given source distribution and a given code, the expected code length is*

$$\bar{\ell} = \mathbb{E}[\ell(X)] = \sum_{x \in \mathcal{X}} p(x)\, \ell(x).$$

Example 13. *Consider the following symbols, their probabilities and their codewords.*

Symbol	***Probability***	***Codeword***
A	1/2	*0*
B	1/4	*10*
C	1/8	*110*
D	1/8	*111*

Thus

$$\bar{\ell} = \sum_{x \in \{A,B,C,D\}} p(x)\ell(x) = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = 1.75.$$

5.2 Intuition Behind Expected Code Length

Suppose the source generates a long sequence $X_1, X_2, \ldots, X_N$. Let $n(x)$ denote the number of times symbol x appears in the sequence, and let $f(x) = n(x)/N$ denote the empirical frequency. Note that the length of the encoded bit sequence for this particular sequence of symbols and a code is

$$\ell(X_1) + \ldots + \ell(X_N) = \sum_{x \in \mathcal{X}} \ell(x)\, n(x).$$

The average number of bits per symbol for this sequence is

$$\frac{\sum_{x \in \mathcal{X}} \ell(x)\, n(x)}{N} = \sum_{x \in \mathcal{X}} \ell(x)\, \frac{n(x)}{N} = \sum_{x \in \mathcal{X}} \ell(x) f(x).$$

For large N (under suitable assumptions), empirical frequencies satisfy $f(x) \approx p(x)$, so

$$\sum_{x \in \mathcal{X}} \ell(x) f(x) \approx \sum_{x \in \mathcal{X}} \ell(x) p(x) = \bar{\ell}.$$

Thus, the expected code length $\bar{\ell}$ characterizes the average number of bits per symbol required to represent a long sequence produced by the source.

Example 14. *For the sequence $ABAAAABBAC$, we have $n(A) = 6$, $n(B) = 3$, $n(C) = 1$, and hence $f(A) = 0.6$, $f(B) = 0.3$, $f(C) = 0.1$. Between the two prefix-free codes*

$$\textit{Code 1: } A \mapsto 00,\ B \mapsto 01,\ C \mapsto 1,$$
$$\textit{Code 2: } A \mapsto 1,\ B \mapsto 01,\ C \mapsto 00,$$

Code 2 assigns the shortest codeword to the most frequent symbol and therefore yields a shorter encoded bitstream.

The example above illustrates a basic principle of efficient compression: symbols that occur more frequently should be assigned shorter codewords, while rare symbols can be assigned longer codewords. This is because frequent symbols contribute more heavily to the total number of bits, and reducing their codeword lengths has the largest impact on the overall average.

6 Huffman Coding

We now turn to the problem of constructing a prefix-free code that minimizes the expected code length $\bar{\ell}$. This is achieved by the Huffman coding algorithm.

6.1 Huffman Coding Algorithm

Input: A set of symbols $\mathcal{X}$ with probabilities $\{p(x)\}_{x\in\mathcal{X}}$.

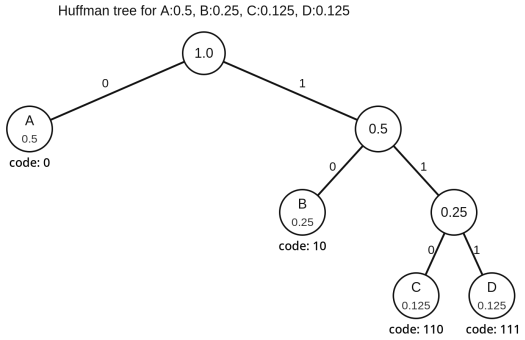
1. **Initialize nodes.** Create a leaf node for each symbol x with probability $p(x)$.
2. **Iteratively merge:** While there is more than one node remaining:
   a. Pick the two nodes with the smallest probabilities.
   b. Merge them into a new parent node whose probability is the sum of the two probabilities.
3. **Form a tree.** The final remaining node is the root of the Huffman tree.
4. **Assign codewords.** Label each left edge as 0 and each right edge as 1 (either convention is fine). The codeword for a symbol is the bit sequence along the path from the root to its leaf.

At a high level, the algorithm repeatedly merges the two least likely symbols. Intuitively, rare symbols should receive long codewords. By merging the smallest probabilities first, the algorithm forces them deeper in the tree. Each merge pushes the combined node one level higher, meaning the original rare symbols accumulate additional bits in their codewords. This greedy strategy turns out to produce the globally optimal tree.

Let us first look at two examples.

Example 15. *Probabilities:*

$$p(A) = 0.5, \quad p(B) = 0.25, \quad p(C) = 0.125, \quad p(D) = 0.125.$$

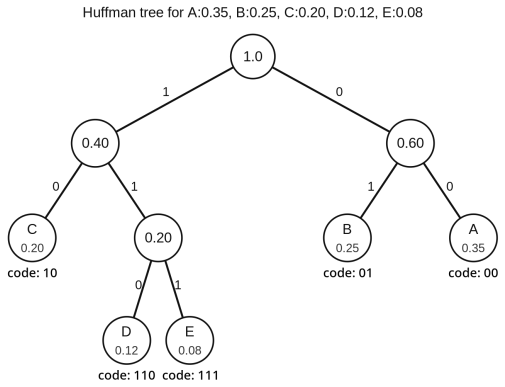


Symbol	Probability	Codeword
A	1/2	0
B	1/4	10
C	1/8	110
D	1/8	111

We computed above that this achieves $\bar{\ell} = 1.75$.

Example 16. *Probabilities:*

$$p(A) = 0.35, \quad p(B) = 0.25, \quad p(C) = 0.20, \quad p(D) = 0.12, \quad p(E) = 0.08.$$



Symbol	Probability	Codeword
A	0.35	00
B	0.25	01
C	0.20	10
D	0.12	110
E	0.08	111

The expected code length is

$$\bar{\ell} = 0.35 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 2 + 0.12 \cdot 3 + 0.08 \cdot 3 = 2.2.$$

We next state an important property about the Huffman coding algorithm. **The output of the Huffman coding algorithm is always a prefix-free code.** This follows directly from the tree construction: codewords are assigned only to leaves, and the codeword for a symbol is the sequence of edge labels along the path from the root to that leaf. If one codeword were a prefix of another, then the leaf corresponding to the first symbol would lie on the path from the root to the second leaf, which would make it an ancestor of that second leaf. But a leaf, by definition, has no children and therefore cannot be an ancestor of another leaf. Hence no codeword can be a prefix of another. Equivalently, during decoding, once we reach a leaf while reading bits, the codeword must terminate since there are no further branches to follow, and therefore it cannot be extended into a longer valid codeword.

The following fact states that the Huffman code is optimal among all prefix-free codes.

Fact 2. *For any source distribution, the Huffman code has the smallest expected code length $\bar{\ell}$ among all prefix-free codes.*

We also have the following theorem, matching our intuition that for an optimal code more frequent symbols have shorter codewords.

Theorem 1. *Let $x, x' \in \mathcal{X}$. If $p(x) < p(x')$, then in a Huffman code we have*

$$\ell(x) \geq \ell(x').$$

Proof. [**OPTIONAL**] Suppose, for contradiction, that $p(x) < p(x')$ but $\ell(x) < \ell(x')$. That is, a less probable symbol is assigned a shorter codeword than a more probable symbol. Intuitively, this is wasteful: since the more probable symbol appears more often, we should prefer giving it the shorter codeword. Consider swapping the codewords assigned to x and x'. The prefix-free property is preserved because we are only relabeling two leaves of the tree; the tree structure itself does not change. After the swap, the more probable symbol x' receives the shorter codeword and the less probable symbol x receives the longer one. The change in expected length is determined only by these two symbols. The difference between the new and old expected lengths is

$$\bar{\ell}' - \bar{\ell} = (p(x) - p(x'))(\ell(x') - \ell(x)).$$

Since $p(x) < p(x')$ and $\ell(x) < \ell(x')$, this quantity is negative, which means the swap strictly reduces the expected code length. This contradicts the optimality of the Huffman code. Therefore it must be that $\ell(x) \geq \ell(x')$ whenever $p(x) < p(x')$. □

Finally, let us look at another example.

Example 17. *Let $\mathcal{X} = \{H, T\}$ with*

$$p(H) = 0.8, \qquad p(T) = 0.2.$$

The Huffman algorithm outputs code with codeword lengths 1 for both H and T. This is true for any distribution with two symbols. So symbol-by-symbol coding yields

$$\bar{\ell} = 1 \text{ bit per symbol.}$$

Even though one symbol is much more likely than the other, we cannot assign fewer than one bit per symbol when coding them individually. This motivates a natural question: can we do anything beyond using this trivial code in this case? Recall that we are often interested in not just encoding a single outcome of the source but a sequence of outcomes. How about, instead of generating a code for each outcome individually, we consider pairs of outcomes together and generate a Huffman code for a block of two outcomes?

7 Block Coding

Instead of encoding one symbol at a time, we encode blocks. We take the following assumption.

Assumption 1. *In our examples, we assume $X_1, X_2, \ldots$ are independent and identically distributed.*

Let us see if this block coding can help for the example we studied before ($P(H) = 0.8$ and $P(T) = 0.2$) by working with blocks of size 2. Consider blocks $X_1X_2 \in \{HH, HT, TH, TT\}$.

Block X_1X_2	**Probability**	**Codeword (Huffman)**
HH	$0.8 \times 0.8 = 0.64$	0
HT	$0.8 \times 0.2 = 0.16$	11
TH	$0.2 \times 0.8 = 0.16$	100
TT	$0.2 \times 0.2 = 0.04$	101

Average bits per block. The expected block code length is

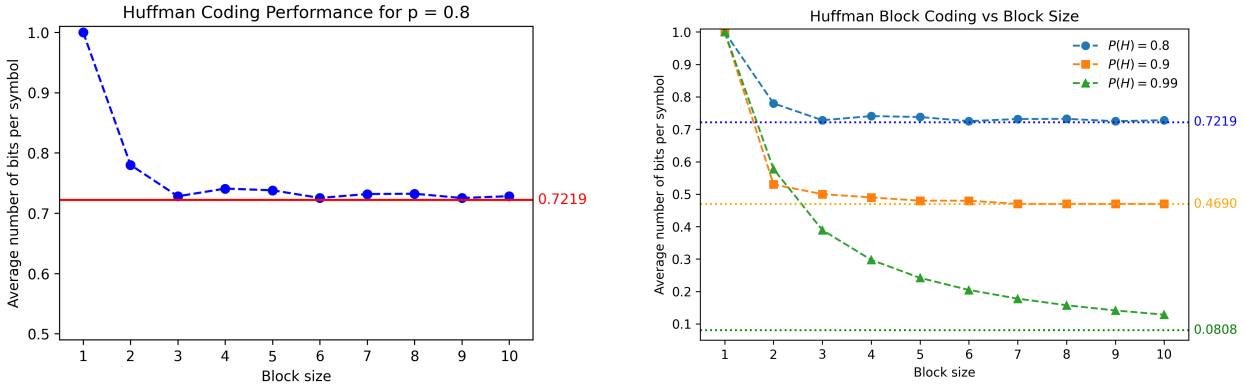
$$\bar{\ell}_{\text{block}} = 0.64 \cdot 1 + 0.16 \cdot 2 + 0.16 \cdot 3 + 0.04 \cdot 3 = 1.56.$$

Average bits per symbol. Since each block contains 2 symbols, the average number of bits per source symbol is

$$\frac{\bar{\ell}_{\text{block}}}{2} = \frac{1.56}{2} = 0.78,$$

which is strictly better than 1 bit/symbol from symbol-by-symbol coding. This shows that we can possibly achieve better coding (i.e., lower average number of bits per symbol) by performing block coding.

The following plots show how the average number of bits per symbol decreases as the block size increases. The left plot corresponds to the skewed binary example discussed above, while the right plot shows the same behavior for three different source distributions.



This raises the following fundamental question which we will discuss next.

> *What are these lower limits on compression? How far can we compress a source, even with large block sizes?*

8 What Is Information?

To answer the fundamental question about lower limits on compression, we first study a quantitative notion of how informative an event is.

8.1 Surprise of an event

Let us think about how informative an event is. If an event is very likely, then learning that it occurred does not tell us much, since this was already expected. On the other hand, if an event is very unlikely, then learning that it occurred is highly informative. In this sense, less likely events are more informative because they are more surprising.

With this intuition, we attempt to define a surprise function S which captures the amount of surprise associated with an event A. We expect the surprise to depend only on the probability of the event, and hence we write the function as $S(p)$ where $p = P(A)$. We impose the following properties on the function $S(p)$:

(1) $S(p)$ is a decreasing function of p, since more likely events are less surprising.

(2) $S(p)$ is a continuous function of p, since we do not expect the surprise to change abruptly for small changes in probability.

(3) Consider two independent events A and B. Intuitively, if two events are independent, then learning that one occurred provides no information about the other, so the total surprise

should simply be the sum of the individual surprises. For example, the surprise should accumulate if you win two independent lotteries. So,

$$S(P(A,B)) = S(P(A)P(B)) = S(P(A)) + S(P(B)).$$

More concisely, we are looking for a function $S(p)$ for $0 < p \leq 1$ which satisfies

$$S(pq) = S(p) + S(q) \quad \text{for all } p, q \in (0,1],$$

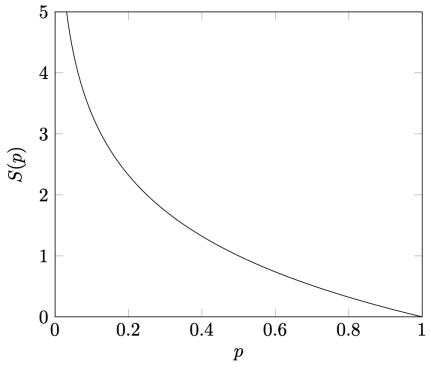
is continuous, and decreases with increasing p.

Fact 3. *The only function satisfying the above properties is*

$$S(p) = \log_2\left(\frac{1}{p}\right),$$

up to a constant multiplicative factor.

Throughout the course, we work with base-2 logarithms, so surprise is measured in bits. The surprise function satisfies several basic properties. First, $S(p) \geq 0$ for all $0 < p \leq 1$, with $S(1) = 0$, since an event that is certain carries no surprise. Moreover, as p decreases, the surprise increases, and in fact $S(p) \to \infty$ as $p \to 0$. This reflects the intuition that extremely unlikely events are arbitrarily surprising. The following figure shows how $S(p)$ varies with p.



8.2 Entropy

Using the notion of surprise, we define entropy as the average surprise of a random variable. Note that event $\{X = x\}$ occurs with probability $P(X = x) = p(x)$ and has surprise $S(P(X = x)) = S(p(x))$.

Definition 10 (Entropy)**.** *Let X be a discrete random variable with alphabet $\mathcal{X}$ and probabilities $p(x)$ for each symbol x. The entropy of X is defined as*

$$H(X) = \sum_{x \in \mathcal{X}} p(x) S(p(x)) = \sum_{x \in \mathcal{X}} p(x) \log_2\left(\frac{1}{p(x)}\right).$$

Entropy of X can be interpreted as the average amount of surprise on learning the value of X. Note that entropy depends only on the probability values and not on the labels of the symbols.

Example 18. *Let X have distribution*

$$p(A) = \tfrac{1}{2}, \quad p(B) = \tfrac{1}{4}, \quad p(C) = \tfrac{1}{8}, \quad p(D) = \tfrac{1}{8}.$$

Then

$$H(X) = \tfrac{1}{2} \cdot 1 + \tfrac{1}{4} \cdot 2 + \tfrac{1}{8} \cdot 3 + \tfrac{1}{8} \cdot 3 = 1.75.$$

Example 19 (Uniform random variable)**.** *Let X be a random variable with a uniform distribution over an alphabet $\mathcal{X}$ with M symbols, i.e.,*

$$p(x) = P(X = x) = \frac{1}{M} \quad \textit{for all } x \in \mathcal{X}.$$

Then its entropy is

$$H(X) = \sum_{x \in \mathcal{X}} p(x) \log_2 \left(\frac{1}{p(x)} \right) = \sum_{x \in \mathcal{X}} \frac{1}{M} \log_2(M) = M \cdot \frac{1}{M} \log_2(M) = \log_2(M).$$

The following fact gives a lower and an upper bound on the entropy of a random variable defined over an alphabet with M symbols.

Fact 4. *If X takes values in an alphabet of size M, then*

$$0 \leq H(X) \leq \log_2 M.$$

Moreover, $H(X) = 0$ if and only if X is deterministic, and $H(X) = \log_2 M$ if and only if X is uniformly distributed.

Note that $H(X) \geq 0$ follows from the fact that $P(X = x) \leq 1$ for all x, and hence $\log_2(1/P(X = x)) \geq 0$. Equality holds if and only if X is deterministic, in which case there is no uncertainty and therefore no surprise associated with the outcome. The above fact also implies that for all source distributions over a given alphabet, the highest entropy is for the uniform distribution.

8.3 Interpretations and Intuition

Entropy can be interpreted in several ways: the randomness or uncertainty in a source, the information content or information value of a source, or the number of bits required to represent a source. Entropy is also very closely connected with how compressible a source is (discussed formally in the next section). There can be many intuitive explanations for how they are connected. We discuss a few of these here.

Entropy can be interpreted as a measure of randomness or uncertainty. For a fixed alphabet size, we saw that the entropy is maximized when the random variable is uniformly distributed. This aligns with the intuitive notion of randomness: a source is most random when every outcome occurs with the same probability.

This observation also connects entropy to compression. When the source distribution is uniform, all symbols occur with the same frequency, and there is little statistical structure to exploit. From our previous discussion, we know that compression relies on assigning shorter codewords to more frequent symbols. When all symbols are equally frequent, this advantage disappears, making the source less compressible on average. In contrast, when the distribution is skewed, some symbols occur much more frequently than others, allowing for shorter codewords and better compression.

Putting these observations together, we arrive at the following informal connection: sources with higher entropy are more random and harder to compress, while sources with lower entropy are more structured and allow more efficient representations. Equivalently, lower entropy corresponds to lower randomness, which corresponds to a more skewed distribution, which in turn allows for greater compressibility and fewer bits required on average to represent the source.

There is also an intuitive link between entropy and representation of information in bits. Recall that k bits can represent 2^k distinct values. For a random variable that is uniformly distributed over M possible outcomes, the entropy is $H(X) = \log_2 M$, which is precisely the number of bits required to represent M equally likely values. As M grows, the entropy increases, reflecting the fact that more bits are needed to represent the outcome of the random variable. This is consistent with our intuitive understanding that a random variable with more possible equally likely outcomes carries more information.

9 Entropy and Huffman Coding

The following fact states that entropy provides a benchmark against which the performance of any lossless coding scheme can be compared.

Fact 5. *For any discrete random variable X, the expected codeword length $\bar{\ell}$ of the Huffman code satisfies*

$$H(X) \leq \bar{\ell} \leq H(X) + 1.$$

The lower bound in the above inequality holds for all prefix-free codes. This follows from the optimality of the Huffman algorithm: among all prefix-free (and more generally, uniquely decodable) codes for a given source distribution, the Huffman code achieves the smallest possible expected codeword length. Since the expected length of the Huffman code is at least the entropy, no other prefix-free code can have an average codeword length smaller than $H(X)$. Thus, the entropy serves as a universal lower bound on the performance of all prefix-free codes.

The upper bound holds only for the Huffman code, showing that Huffman coding performs close to the benchmark of entropy, with a gap of at most one bit.

Example 20. *Consider the first example of Huffman code from the previous lecture.*

Symbol	***Probability***	***Huffman Code***
A	1/2	*0*
B	1/4	*10*
C	1/8	*110*
D	1/8	*111*

The average code length of this code is 1.75*. We also verified earlier in this lecture that the entropy of this source is* 1.75*.*

So, in the above example, the expected codeword length of the Huffman code exactly matches the entropy. This is not a coincidence. It illustrates a special class of source distributions for which Huffman coding achieves entropy exactly.

Dyadic distributions A source is said to have a *dyadic distribution* if all symbol probabilities are negative powers of two:

$$P(X = x) = 2^{-k_x}$$

for some positive integers k_x. For such distributions, the Huffman algorithm produces a prefix-free code whose codeword lengths satisfy

$$\ell(x) = \log_2\left(\frac{1}{P(X = x)}\right),$$

and consequently,

$$\bar{\ell} = H(X).$$

Intuitively, dyadic distributions align perfectly with binary representations: probabilities that are exact powers of two correspond to integer codeword lengths. As a result, there is no inefficiency introduced by rounding codeword lengths to integers, and the entropy can be achieved exactly.

Skewed Distributions. Now, we discuss another example where the distribution is very skewed.

Example 21. *Consider a binary random variable X with $P(H) = 0.99$, and $P(T) = 0.01$. The entropy of this source is* 0.0808*. As discussed in the previous lecture, the Huffman code assigns one bit to both symbols, and hence $\bar{\ell} = 1$. In this example, the gap between $\bar{\ell}$ and $H(X)$ is large relative to the entropy itself.*

This example highlights a limitation of symbol-by-symbol coding. To understand how much improvement is fundamentally possible, we now analyze block coding more carefully and relate its performance to entropy. This will allow us to characterize the minimum achievable average number of bits per symbol.

10 Joint Entropy and Block Coding

When symbols are encoded in blocks, the block itself can be viewed as a single random variable whose distribution is determined by the joint distribution of the individual symbols.

10.1 Joint entropy

The joint entropy of two random variables denotes the information the two r.v.s together contain.

Definition 11 (Joint entropy)**.** *Let X and Y be discrete random variables with alphabets $\mathcal{X}$ and $\mathcal{Y}$, respectively. The joint entropy of (X, Y) is defined as*

$$H(X, Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(X = x, Y = y) \log_2\left(\frac{1}{P(X = x, Y = y)}\right).$$

This definition can be interpreted by treating the pair (X, Y) as a single "super-symbol" that takes values in the product alphabet $\mathcal{X} \times \mathcal{Y}$. The joint entropy then measures the average surprise associated with observing the pair (X, Y).

Independent random variables

When X and Y are independent, each random variable has its own randomness or uncertainty, and knowing the value of one does not reduce the uncertainty of the other. As a result, the total information contained in the pair (X, Y) is simply the sum of the information contained in X and the information contained in Y. Since entropy measures this uncertainty, the joint entropy of two independent random variables equals the sum of their individual entropies. We state this formally in the following theorem.

Theorem 2. *For any pair of independent random variables X and Y,*

$$H(X, Y) = H(X) + H(Y).$$

Proof. Recall that if X and Y are independent, then

$$P(X = x, Y = y) = P(X = x)P(Y = y) \quad \text{for all } x \in \mathcal{X},\ y \in \mathcal{Y}.$$

Therefore,

$$\begin{aligned}
H(X,Y) &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(X=x,Y=y)\log_2\left(\frac{1}{P(X=x,Y=y)}\right)\\
&= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(X=x)P(Y=y)\log_2\left(\frac{1}{P(X=x)P(Y=y)}\right)\\
&= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(X=x)P(Y=y)\left(\log_2\frac{1}{P(X=x)}+\log_2\frac{1}{P(Y=y)}\right)\\
&= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(X=x)P(Y=y)\log_2\frac{1}{P(X=x)} + \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} P(X=x)P(Y=y)\log_2\frac{1}{P(Y=y)}\\
&= \sum_{x\in\mathcal{X}} P(X=x)\log_2\frac{1}{P(X=x)}\left(\sum_{y\in\mathcal{Y}} P(Y=y)\right)\\
&\qquad\qquad + \sum_{y\in\mathcal{Y}} P(Y=y)\log_2\frac{1}{P(Y=y)}\left(\sum_{x\in\mathcal{X}} P(X=x)\right)\\
&= \sum_{x\in\mathcal{X}} P(X=x)\log_2\frac{1}{P(X=x)} + \sum_{y\in\mathcal{Y}} P(Y=y)\log_2\frac{1}{P(Y=y)}\\
&= H(X) + H(Y),
\end{aligned}$$

where we used $\sum_x P(X = x) = \sum_y P(Y = y) = 1$. □

General Case

When two random variables are not independent, they share some information. In this case, knowing the value of one random variable reduces the uncertainty about the other. As a result, the total information contained in the pair (X, Y) is smaller than the sum of the information contained in X and Y individually. The following fact states this formally.

Fact 6. *For any two random variables X and Y,*

$$H(X,Y) \leq H(X) + H(Y),$$

with equality if and only if X and Y are independent.

10.2 Block Coding

Assume that $X_1, X_2, \ldots$ are independent and identically distributed copies of a random variable X. Suppose we apply the Huffman algorithm to blocks of size n, i.e., we treat the block $(X_1, \ldots, X_n)$ as one "super symbol". Let the average codeword length of the Huffman code for this block source be $\bar{\ell}_{\text{block},n}$ (measured in bits per block). Since Huffman coding is optimal for the block distribution, its expected block length is bounded in terms of the entropy of the block:

$$H(X_1, \ldots, X_n) \;\leq\; \bar{\ell}_{\text{block},n} \;\leq\; H(X_1, \ldots, X_n) + 1.$$

Because $X_1, \ldots, X_n$ are independent and identically distributed, their joint entropy satisfies

$$H(X_1, \ldots, X_n) = H(X_1) + \cdots + H(X_n) = nH(X_1).$$

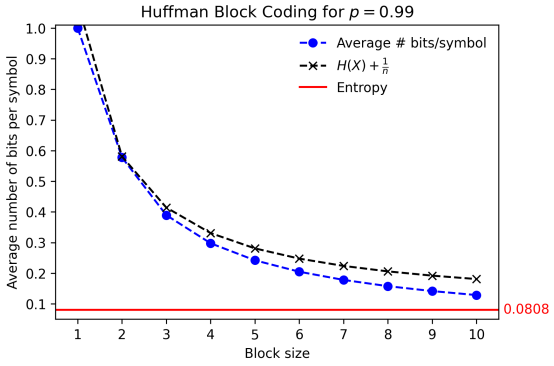
Substituting this into the previous inequality gives

$$nH(X_1) \;\leq\; \bar{\ell}_{\text{block},n} \;\leq\; nH(X_1) + 1.$$

To convert from bits per block to bits per symbol, we divide by the block length n. Thus, the average number of bits per symbol is $\bar{\ell}_{\text{block},n}/n$, and dividing the inequality above by n yields

$$H(X_1) \;\leq\; \text{Average \# bits per symbol} \;=\; \frac{\bar{\ell}_{\text{block},n}}{n} \;\leq\; H(X_1) + \frac{1}{n}.$$

This shows that the gap between the average number of bits per symbol and the entropy shrinks as the block length n increases. In particular, by taking n large, we can make the average number of bits per symbol arbitrarily close to $H(X_1)$. The following plot shows this for the example with the skewed distribution we studied above.



11 Shannon's Source Coding Theorem

We now state the main result that formalizes the connection between entropy and lossless compression. From the block coding bounds derived above, we saw that by encoding larger blocks of symbols, the average number of bits per symbol approaches entropy. This is precisely what Shannon's Source Coding Theorem states.

Theorem 3 (Shannon's Source Coding Theorem)**.** *The entropy of a source equals the minimum number of bits per source symbol necessary on average to encode a sequence of independent and identically distributed symbols from that source. In general, this may require the use of block coding, where blocks of symbols are encoded together.*

The above theorem shows how entropy is not just an abstract measure of information, but a fundamental limit on lossless compression as well. It is not only a lower limit on compression, but also a limit that can be achieved by coding blocks of symbols using the Huffman algorithm. Huffman coding provides an explicit and efficient method for constructing optimal prefix-free codes, which also allow for efficient decoding of bit sequences. Block coding then allows us to use Huffman code to approach the entropy of an i.i.d. source arbitrarily closely.

12 Dependent Sources and Exploiting Structure

So far, we have focused on sources where symbols are independent and identically distributed. In this setting, entropy $H(X)$ characterizes the fundamental limit on lossless compression in terms of bits per symbol. Many real-world sources, however, exhibit strong *dependencies* across symbols. Examples include text, audio, images, and video, where consecutive symbols are often highly correlated.

When a source is dependent, knowing past symbols provides information about future symbols. As a result, the joint entropy of a block of symbols can be *strictly smaller* than the sum of their individual entropies. Formally, for a dependent source,

$$H(X_1, \ldots, X_n) < H(X_1) + \cdots + H(X_n) = nH(X_1).$$

This means that the average information per symbol in a block is *lower* than the entropy of a single symbol.

12.1 Block Coding

Applying block coding to a dependent source allows us to exploit this reduction in joint entropy. Since Huffman coding can approach the joint entropy of the block, the average number of bits per symbol can be made close to

$$\frac{1}{n} H(X_1, \ldots, X_n),$$

which can be strictly smaller than $H(X_1)$. In this sense, for dependent sources, we can achieve compression rates that are *better than the entropy of a single symbol*. Block coding is one way to exploit dependence, but it is not the only way. In practice, it is often beneficial to first transform the data into a representation where the dependence is reduced or simplified, and then apply standard coding techniques.

12.2 Delta Coding

A simple and widely used example of exploiting dependence is *delta coding*. Instead of encoding the sequence $(X_1, X_2, X_3, \dots)$ directly, we encode

$$X_1,\ X_2 - X_1,\ X_3 - X_2,\ X_4 - X_3,\ \dots$$

For many sources of interest, such as smooth signals, images, or video, consecutive values are close to each other. As a result, the differences (or *deltas*) tend to have a much more concentrated distribution than the original values. This leads to lower entropy and hence better compression. Delta coding does not remove information; it simply represents the same information in a different form, one that exposes structure and reduces redundancy.

Example 22 (Delta coding)**.** *Consider a sequence of symbols $X_1, X_2, X_3, \dots$ with the special structure*

$$X_{i+1} = X_i + Z_i,$$

where $\{Z_i\}$ are independent random variables taking values in

$$\{-1, 0, 1\} \quad \textit{with} \quad P(Z_i = -1) = P(Z_i = 0) = P(Z_i = 1) = \tfrac{1}{3}.$$

An example realization of such a sequence is

$$480,\ 479,\ 478,\ 479,\ 479,\ 480,\ 480,\ 481,\ 480,\ \dots$$

Instead of encoding the original sequence directly, we store the first value X_1 and then encode the consecutive differences:

$$X_2 - X_1,\ X_3 - X_2,\ X_4 - X_3,\ \dots$$

For the sequence above, this produces

$$-1,\ -1,\ 1,\ 0,\ 1,\ 0,\ 1,\ -1,\ \dots$$

These differences are independent and identically distributed, take values from a small alphabet, and have significantly lower entropy than the original symbols. As a result, they can be compressed much more efficiently using standard lossless coding techniques.

Delta coding is not just a theoretical idea. It is used in practical compression systems as well. For example, the PNG image compression format applies a form of delta coding as its very first step. Instead of storing raw pixel values, PNG predicts the value of each pixel using neighboring pixels (typically the pixel to the left, above, or a simple combination of both) and then stores the difference between the actual pixel value and this prediction. These prediction errors tend to be small and concentrated around zero, leading to lower entropy and more efficient compression. Subsequent stages of PNG compression then encode these differences using standard lossless coding techniques.

Representation matters. What delta coding illustrates more broadly is a key principle of compression: *the choice of representation is crucial*. The same underlying information can be represented in many different ways, but some representations make the structure in the data much easier to exploit for compression than others. In the next lecture, we will study a particularly powerful class of representations that play a central role in modern signal and image compression: *frequency domain representations*.